# 字符串处理

## 分割（Splitter）

JDK内建的字符串拆分工具（split）有一些古怪的特性，绝对不要用。比如，它会悄悄丢弃了尾部的分隔符。

Splitter强大安全而且是链式编程。

比如，要分割如下字符串：alex,,45, tianshan road.,201301

* 一般用法

|  |
| --- |
| Splitter.on(',')  .trimResults() //移除结果字符串的前导空白和尾部空白  .omitEmptyStrings() //从结果中自动忽略空字符串  .limit(2) //限制拆分出的字符串数量  .split("alex,,45, tianshan road.,201301"); |

以上代码，看着就舒服，而且还不需要产生很多中间变量。

* 使用正则表达式来描述分隔符

|  |
| --- |
| Splitter.onPattern("\\s+")  .split("1 \t 2 3"); //结果为：["1", "2", "3"] |

* 根据长度来拆分字符串

|  |
| --- |
| Splitter. fixedLength(3)  .split("1 2 3"); //结果为：["1 2", " 3"] |

拆分器返回的是Iterable<String>。

如果想直接返回List，只要使用Lists.newArrayList(splitter.split(string))或类似方法。

**重点：splitter实例总是不可变的。**

**用来定义splitter目标语义的配置方法总会返回一个新的splitter实例。这使得splitter实例都是线程安全的，你可以将其定义为static final常量。（也就是说，可以把split函数之前的代码定义成一个全局静态变量，到处使用）。**

## 连接（Joiner）

* 一般用法

|  |
| --- |
| Joiner joiner = Joiner.on(delimiter).skipNulls();  String res = joiner.join("123", null, "foo","bar"); |

* appendTo到实现了Appendable接口的类中

|  |
| --- |
| Joiner joiner = Joiner.on(delimiter).skipNulls();  // 1：append到StringBuilder  StringBuilder stringBuilder = new StringBuilder();  joiner.appendTo(stringBuilder, " foo ", " bar ");  // 2：append到输出流  FileWriter writer = new FileWriter("append\_text.txt");  joiner.appendTo(writer, " foo ", " bar ");  writer.close(); |

## 字符匹配处理（CharMatcher）

它是一个简化版正则工具。

* 按照定义的匹配规则，对字符串进行替换、保留、移除等操作

|  |
| --- |
| String lettersAndNumbers = "foo989yxbar234";  String retained =  CharMatcher.JAVA\_DIGIT.retainFrom(lettersAndNumbers); //保留数字  String removed =  CharMatcher.JAVA\_DIGIT.removeFrom(lettersAndNumbers); //移除数字  //通过Or组合多个匹配规则，再进行操作  CharMatcher cm = CharMatcher.JAVA\_DIGIT.or(CharMatcher.WHITESPACE);  String retained = cm.retainFrom("foo9 89y xbar 234");//保留数字和空格 |

**CharMatcher的用法总结：**

1. 定义匹配字符的规则
2. 使用内置的函数对字符进行处理（移除、保留等）

各处理函数：

* collapseFrom：连续的匹配字符替换为特定字符。比如把多个空格替换成一个
* removeFrom：移除所有匹配字符
* retainFrom：保留匹配字符，移除其他字符
* trimFrom：移除字符序列的前导匹配字符和尾部匹配字符。
* replaceFrom：用特定字符序列替代匹配字符
* matchesAllOf：测试是否字符序列中的所有字符都匹配

# 集合

## 不可变集合

不可变集合是不可被修改的集合。集合的数据项是在创建的时候提供，并且在整个生命周期中都不可改变。

不可变集合是极其有益的一种存储数据的形式，请仔细思考你的程序，把一次创建多次读取的数据用不可变集合来承载。好处是：

* 不可变集合不需要考虑变化，因此可以节省时间和空间。所有不可变的集合都比它们的可变形式有更好的内存利用率
* 不可变对象被多个线程调用时，不存在竞态条件问题
* 不可变对象因为有固定不变，可以作为常量来安全使用
* 创建对象的不可变拷贝是一项很好的防御性编程技巧

**重要提示：Guava不可变集合的实现都不接受null值。**

### 怎么使用不可变集合

不可变集合可以用如下多种方式创建：

1. copyOf方法，如ImmutableSet.copyOf(set);
2. of方法，如ImmutableMap.of(“a”, 1, “b”, 2);
3. Builder工具

|  |
| --- |
| public static final ImmutableSet<Color> GOOGLE\_COLORS =  ImmutableSet.<Color>builder()  .addAll(WEBSAFE\_COLORS)  .add(new Color(0, 191, 255))  .build(); |

对有序不可变集合来说，排序是在构造集合的时候完成的：

ImmutableSortedSet.of("a", "b", "c", "a", "d", "b");

//会在构造时就把元素排序为a, b, c, d

### 智能的copyOf

|  |
| --- |
| List<String> list=new ArrayList<String>();  list.add("one");  list.add("two");  list.add("three");  //用上面这个list构造一个不可变集合：  ImmutableList<String> imList=ImmutableList.copyOf(list); |

这样就用可变List对象中的数据，"copy"出来了一份不可变List。以后，对原List的增删改，都不影响imList。

copyOf方法比你想象的要智能，ImmutableXXX.copyOf会在合适的情况下避免拷贝元素的操作。

### 不可变集合都包括哪些

ImmutableList、ImmutableSet、ImmutableSortedSet、ImmutableMap、ImmutableSortedMap、ImmutableMultiset、ImmutableSortedMultiset、ImmutableMultimap、ImmutableListMultimap、ImmutableSetMultimap、ImmutableBiMap、ImmutableTable、ImmutableClassToInstanceMap、ImmutableCollection

## 新的好用的集合

### Multiset

Multiset和Set的区别就是可以保存多个相同的对象，并且对相同的对象自动计数。

比如，统计一个词在文档中出现了多少次，传统的做法是这样的：

|  |
| --- |
| Map<String, Integer> counts = new HashMap<String, Integer>();  for (String word : words) {  Integer count = counts.get(word);  if (count == null) {  counts.put(word, 1);  } else {  counts.put(word, count + 1);  }  } |

**如果使用Multiset，两行代码就搞定了：**

Multiset<String> wordsMultiset = HashMultiset.create();

wordsMultiset.addAll(words);

### Multimap

在日常的开发工作中，我们有的时候需要构造像Map<K, List<V>>或者Map<K, Set<V>>这样比较复杂的集合类型的数据结构，以便做相应的业务逻辑处理。这种代码在编写、阅读、修改等各个方面都是极其痛苦的。

比如，我们想在Map里面以姓名为KEY，把学生的每门课的成绩保存起来，代码会写成这样：

|  |
| --- |
| Map<String, List<StudentScore>> StudentScoreMap = new HashMap<String, List<StudentScore>>();  List<StudentScore> listScore = new ArrayList< StudentScore>();  listScore.add(StudentScore1);  listScore.add(StudentScore2);  StudentScoreMap.put("peter", listScore); |

这种代码，每次要增加/修改数据时，都要在脑子里面从Map到List切换，会晕死的。使用Multmap，这个世界瞬间就清净了：

|  |
| --- |
| Multimap<String, StudentScore> scoreMultimap  = ArrayListMultimap.create();  scoreMultimap.put("peter", StudentScore1);  scoreMultimap.put("peter", StudentScore2); |

ListMultimap.get(key)返回List，SetMultimap.get(key)返回Set。

注意：Multimap.get(key) 一定返回一个非null的集合。也就是说，如果KEY不存在，也不会返回null，而是空集合。这种设计真TM好。

### BiMap（键值双向映射的Map）

传统上，实现键值对的双向映射需要维护两个单独的map，并保持它们间的同步。但这种方式很容易出错，而且对于值已经在map中的情况，会变得非常混乱。例如：

|  |
| --- |
| Map<String, Integer> nameToId = Maps.newHashMap();  Map<Integer, String> idToName = Maps.newHashMap();  nameToId.put("Bob", 42);  idToName.put(42, "Bob");  //为了反向取值，要使用第2个Map。如果代码有几百行，谁还能记住这个Map  idToName.get(42); |

这种程序很难维护，比如，你要时刻记得同时维护两个Map，nameToId中增加了值，要记得再往idToName里面反向增加对应的值。明明数据在nameToId里面存着，为了方向找数据，还要跑去另外一个Map里面取值，这种代码实在是恶心。

BiMap的使用方式：

|  |
| --- |
| BiMap<String, Integer> user\_id = HashBiMap.create();  user\_id.put("Bob", 42);  //反向取值，仍然使用这个Map就可以  String userName = user\_id.inverse().get(42);  BiMap<Integer, String > id\_user = user\_id.inverse();  id\_user.put(58, "alex");//原user\_id这个map同步生效 |

而且，对于反转后的map的所有操作天然对原map生效，完全避免了原来那种编程方式导致的两个map不一致的可能性。

### Table

当我们需要多个索引的数据结构的时候，通常情况下，我们只能用这种丑陋的Map<FirstName, Map<LastName, Person>>来实现，或者使用更恶心的二维数组。

为此Guava提供了一个新的集合类型－Table集合类型，来支持这种数据结构的使用场景。Table支持“row”和“column”的方式进行数据操作。

|  |
| --- |
| Table<Vertex, Vertex, Double> table = HashBasedTable.create();  table.put(v1, v2, 4);  table.put(v1, v3, 20);  table.put(v2, v3, 5);  table.row(v1); // returns a Map mapping v2 to 4, v3 to 20  table.column(v3); // returns a Map mapping v1 to 20, v2 to 5 |

## 增强了功能的集合

与JDK内置的各个集合对应着提供了更强功能集合。都是后面加了个s，比如：Lists、Maps、Sets等，以及与自身新集合对应的增强集合，比如：Multisets、Tables等等。

1. **更易于创建**

构造范型集合时要讨厌地重复声明范型：

List<String> list = new ArrayList<String>();

用Guava就简单了：List<String> list = Lists.newArrayList();

方便地在初始化时就指定起始元素：

List<String> theseElements = Lists.newArrayList("alpha", "beta", "gamma");

1. **更多有用的方法**

* 使用Lists.partition()方法分割列表：

List<List<String>> subList = Lists.partition(ps, 2);

// [a, b, c, d, e] 分割成两个： [[a, b], [c, d, e]]

* Sets.difference()求S1-S2：

Set<String> s1 = Sets.newHashSet("1", "2", "3");

Set<String> s2 = Sets.newHashSet("2", "3", "4");

Sets.difference(s1, s2); //[1]

* Sets. intersection()求S1,S2交集：

Set<String> s1 = Sets.newHashSet("1", "2", "3");

Set<String> s2 = Sets.newHashSet("2", "3", "4");

Sets.SetView<String> sv = Sets.intersection(s1, s2); // [2, 3]

* Sets. union()求合集：

Set<String> s1 = Sets.newHashSet("1", "2", "3");

Set<String> s2 = Sets.newHashSet("2", "3", "4");

Sets.SetView<String> sv = Sets.union(s1, s2); // [3, 2, 1 ,4]

# 缓存

缓存的用途：计算或检索一个值的代价很高，并且对同样的输入需要不止一次获取值的时候，就应当考虑使用缓存。

## 概览

Guava Cache与ConcurrentMap很相似，但也不完全一样。最基本的区别是ConcurrentMap会一直保存所有添加的元素，直到显式地移除。相对地，Guava Cache为了限制内存占用，通常都设定为自动回收元素。在某些场景下，尽管LoadingCache 不回收元素，它也是很有用的，因为它会自动加载缓存。

Guava Cache适用于：

* + 你愿意消耗一些内存空间来提升速度
  + 你预料到某些键会被查询一次以上
  + 缓存中存放的数据总量不会超出内存容量
  + 希望对缓存的数据能够有效的控制，并避免繁琐的编程

Guava Cache是单个应用运行时的本地缓存。它不把数据存放到文件或外部服务器。如果这不符合你的需求，应使用Memcached这类工具

看看Guava Cache的用法例子：

|  |
| --- |
| LoadingCache<Key, Graph> graphs = CacheBuilder.newBuilder()  .maximumSize(1000) //设置容量  .expireAfterWrite(10, TimeUnit.MINUTES) //设置过期时间  .removalListener(MY\_LISTENER) //设置处理函数  .build(  new CacheLoader<Key, Graph>() {  public Graph load(Key key) throws AnyException {  return createExpensiveGraph(key);  }  }); |

## 缓存数据的回收

Guava Cache提供了三种基本的缓存回收方式：基于容量回收、定时回收和基于引用回收。

* 基于容量回收

如果要规定缓存项的数目不超过固定值，只需使用CacheBuilder.maximumSize(long)。缓存将尝试回收最近没有使用或总体上很少使用的缓存项。

在缓存项的数目达到限定值之前，缓存就可能进行回收操作——通常来说，这种情况发生在缓存项的数目逼近限定值时。

另外，不同的缓存项有不同的“权重”（weights），如果你的缓存值，占据完全不同的内存空间，你可以使用CacheBuilder.weigher(Weigher)指定一个权重函数，并且用CacheBuilder.maximumWeight(long)指定最大总重。

* 定时回收

1）缓存项在给定时间内没有被读/写访问，则回收

2）缓存项在给定时间内没有被写访问（创建或覆盖），则回收。

* 基于引用的回收

通过使用弱引用的键、或弱引用的值、或软引用的值，Guava Cache可以把缓存设置为允许垃圾回收。

## 缓存数据清除

* 手工显式清除

任何时候，你都可以显式地清除缓存项，而不是等到它被回收：

* + 1. 个别清除：Cache.invalidate(key)
    2. 批量清除：Cache.invalidateAll(keys)
    3. 清除所有缓存项：Cache.invalidateAll()
* 使用移除监听器

通过CacheBuilder.removalListener(RemovalListener)，你可以声明一个监听器，以便缓存项被移除时做一些额外操作。缓存项被移除时，RemovalListener会获取移除通知[RemovalNotification]，其中包含移除原因[RemovalCause]、键和值。

## 刷新

刷新和回收不太一样。正如LoadingCache.refresh(K)所声明，刷新表示为键加载新值，这个过程可以是异步的。在刷新操作进行时，缓存仍然可以向其他线程返回旧值，而不像回收操作，读缓存的线程必须等待新值加载完成。

可以为缓存增加自动定时刷新功能。和expireAfterWrite相反，refreshAfterWrite通过定时刷新可以让缓存项保持可用，但请注意：缓存项只有在被检索时才会真正刷新（如果CacheLoader.refresh实现为异步，那么检索不会被刷新拖慢）。因此，如果你在缓存上同时声明expireAfterWrite和refreshAfterWrite，缓存并不会因为刷新盲目地定时重置，如果缓存项没有被检索，那刷新就不会真的发生，缓存项在过期时间后也变得可以回收。

## 其他特性

CacheBuilder.recordStats()用来开启Guava Cache的统计功能。统计打开后，Cache.stats()方法会返回CacheStats对象以提供如下统计信息：

* hitRate()：缓存命中率；
* averageLoadPenalty()：加载新值的平均时间，单位为纳秒；
* evictionCount()：缓存项被回收的总数，不包括显式清除。

# 哈希

## 概览

Hashing类提供了若干散列函数，以及运算HashCode对象的工具方法。

* 已提供的散列函数

md5() murmur3\_128() murmur3\_32() orc32()

sha1()/sha256()/sha512() adler32() sipHash24() goodFastHash(int bits)

* 用法示例：

|  |
| --- |
| HashFunction hf = Hashing.md5();  HashCode hc = hf.newHasher()  .putLong(id)  .putString(name, Charsets.UTF\_8)  .hash(); |

* 文件哈希

File file = new File("src/main/resources/sampleTextFileOne.txt");

HashCode hashCode = Files.hash(file, Hashing.md5());

## 关于MurmurHash

这东西在Redis，Memcached，Cassandra，HBase，Lucene里面都在使用。是一个简单高效且还是分布式的算法

关于Hash，我们之前只知道MD5，SHA1，SHA256还有Java自己的hashCode()，而MurmurHash这算法是2008年才被发明的，与MD5这些讲究安全性的摘要算法比，Redis们内部为主键做个Hash而已，就不需要安全性了，因此Google家的MurmurHash这种non-cryptographic的速度会快几十倍。

【参考文章：<http://www.oschina.net/translate/state-of-hash-functions】>

# 数学运算

使用GUAVA的数据运算工具的好处：

* Guava Math针对各种不常见的溢出情况都有充分的测试
* Guava Math的性能经过了精心的设计和调优。某些场景下甚至优于Apache Commons的MathUtils
* 其工具函数的命名和用法，可读性高

其包含了大量的预算工具，包括：

* 整数运算，且进行了溢出检查
* 实数运算，包括：除法、2为底的对数、10为底的对数、平方根，提供各种舍入处理
* 各种数学运算，包括：最大公约数、取模、取幂、是否2的幂、阶乘、二项式系数

对于浮点数运算，直接使用JDK自带的功能函数即可。

# 其他有益工具

## 如何使用或避免使用null

在程序中，很多情况下，使用null以后，会导致代码意义混乱！比如：

1. 情况一：如果存入Map的数据为null，那么，后续程序从该Map中取值时，你就不知道是因为key不存在返回了null，还是这个key在Map存了一个null。
2. 很多人在设计有返回值的方法时，会把返回null作为该方法执行成功或失败。但是这种设计会让调用者的代码看起来极其丑陋。

总之，你的代码里面就不应该存在类似**if( XXX==null )**这样判断逻辑！

**Guava的Optional：**

* 如果你的方法的返回值是String，并且，在该方法中，某些情况下，你准备返回null,那么，你应该把该方法的返回值设计为**Optional<String>**。把返回null，改成返回**Optional.absent()**。

在调用这个方法的程序中，使用isPresent()函数代替“XXX==null”，来判断返回值。

* 如果你存到Map里面的值有可能为null，那么应该这样写：

|  |
| --- |
| String value = ……;  **Optional<String> mapValue = Optional.fromNullable(value);**  **Map<String, Optional<String>> map**  **= new HashMap<String, Optional<String>>();**  map.put(“name”, mapValue); |

这种方式，在你用get()取值后，再用**if( XXX==null )，**你就可以明确知道是Map中不存在get()函数中的key，还是这个key对应的值为null。

* Optional的可用方法：
  1. Optional.of(T)：初始化Optional对象。如果T为null，则抛出异常
  2. Optional.absent()：获得一个意义为“缺失”的Optional对象（也就是null）
  3. Optional.fromNullable(T)：初始化一个可为null的对象。如果T为null，则等价于absent()。
  4. boolean isPresent()：判断Optional对象是否存在（是否为null）
  5. T get()：返回Optional包含的对象的值。该对象的值不是为null，否则抛异常
  6. T orNull()：返回Optional包含的对象的值，如果只为null，则返回null。
  7. T or(T)：有默认值的返回。

Optional对象的使用，强迫你去积极的思考该怎样处理你的代码中的null！

这种思考会在返回某些存在的值或者不存在的值的时候显得特别相关。

和其他人一样，你绝对很可能会忘记别人写的方法method(a,b)可能会返回一个null值，就好像当你去写method(a,b)的实现时，你也很可能忘记输入参数a也可以是null。

如果返回的是Optional对象，对于调用者来说，就可以忘却怎么去度量null代表的是什么含义，因为他们始终要从optional对象中去获得真正的返回值。

## 易用的比较接口

我们经常需要对两个对象进行比较，为此，必须实现compareTo方法，并且在方法中通过大量的if/else对各个属性值进行比较，类似下面这样：

|  |
| --- |
| class Girl implements Comparable<Girl> {  private String name;//姓名  private int height;//身高  //传统方法我们这样比较  public int compareTo(Girl girl) {  int c1 = name.compareTo(girl.getName());  if (c1 != 0){  return c1;  }  if (height>girl.getHeight()){  return 1;  }  else if(height<girl.getHeight()){  return -1;  }  return 0;  }  }//**实际上，我非常讨厌上面这些1/-1/0，因为我根本搞不清楚返回1是大还是小** |

现在，看看GUAVA里面是怎么写的：

|  |
| --- |
| public int compareTo(Girl girl) {  return **ComparisonChain**.start() //链式比较,在第一个非0处返回  .compare(this.name, girl.getName())  .compare(this.height, girl.getHeight ())  .result();  }  } |

当我们在任何需要自定义比较器的时候，可以ComparisonChain来快速且简单的实现比较逻辑：我去比我想比的变量就可以了，根本不用去管什么if/else，以及返回1与-1之间的意义。

在compare里面还可以传入更复杂的比较器，比如：

|  |
| --- |
| final Comparator<String> strLengthComparator = new Comparator<String>(){  @Override  public int compare(String str1, String str2) {  return str2.length() - str1.length();  }};  //上面的比较使用这个比较器：  public int compareTo(Girl girl) {  return **ComparisonChain**.start() //链式比较,在第一个非0处返回  .compare(this.name, girl.getName(),  **Ordering.from(strLengthComparator)**)  .compare(this.height, girl.getHeight())  .result();  }  } |

# 网址

## <http://blog.csdn.net/yyychyzzzz/article/details/54983574> Google guava工具类的介绍和使用

## <http://wiki.jikexueyuan.com/project/google-guava-official-tutorial/primitives.html> 官方翻译文档

## <https://github.com/google/guava/wiki> 官方wiki地址